Serdica J. Computing **11** (2017), No 1, 45–57

Serdica
Journal of Computing

Bulgarian Academy of Sciences
Institute of Mathematics and Informatics

FAST BITWISE IMPLEMENTATION OF THE ALGEBRAIC NORMAL FORM TRANSFORM[*]

Valentin Bakoev

Abstract. The representation of Boolean functions by their algebraic normal forms (ANFs) is very important for cryptography, coding theory and other scientific areas. The ANFs are used in computing the algebraic degree of S-boxes, some other cryptographic criteria and parameters of error-correcting codes. Their applications require these criteria and parameters to be computed by fast algorithms. Hence the corresponding ANFs should also be obtained by fast algorithms. Here we continue our previous work on fast computing of the ANFs of Boolean functions. We present and investigate the full version of bitwise implementation of the ANF transform. When we use a bitwise representation of Boolean functions in 64-bit computer words, we obtain a time complexity of $\Theta((n+44)\ .\ 2^{n-7})$ and a space complexity of $\Theta(2^{n-6})$. The experimental results show that this implementation is more than 25 times faster in comparison to the well-known byte-wise ANF transform.

ACM Computing Classification System (1998): F.2.1, F.2.2.

Key words: Boolean function, algebraic normal form transform, Möbius (Moebius) transform, Zhegalkin transform, positive polarity Reed–Muller transform, bitwise implementation.

[*]This work was partly supported by the Research Fund of the University of Veliko Turnovo (Bulgaria) under Contract FSD-31-653-07/19.06.2017. Some of the results were announced at the OCRT mini-symposium in the frames of the Mathematics Days in Sofia, 10–14 July, 2017.

1. Introduction.

The representation of a Boolean function by its Algebraic Normal Form (ANF) is widely used in cryptography, coding theory (in Reed–Muller codes) and many other scientific areas [3, 4]. Each of them involves Boolean functions that satisfy specific criteria. Some of them are defined and computed by the corresponding Algebraic Normal Forms (ANFs). For example, the algebraic degree of a vectorial Boolean function (called also an S-box) is one of its most important cryptographic parameters. Its computing operates with the coordinate Boolean functions that form the S-box and is done in two main ways:

- by obtaining the ANFs of these functions and operating with them;
- by operating with the truth table vectors of the functions. In [6] some algorithms of such type are developed, but their complexities are not derived. This approach is quite faster than the first one for about the half of all Boolean functions of n variables. However the numerical results do not show the superiority of this approach for the remaining Boolean functions. These results raise some questions, discussed at the end of this work.

In generation of S-boxes the computing of the algebraic degree should be done as fast as possible (as well as the computing of the other cryptographic criteria), in order to obtain more S-boxes and to select the best of them. This requires fast computing of the ANFs of the coordinate Boolean functions. A need for efficient computing of the ANFs or algebraic degrees of Boolean functions arises in many other cases, for example in computing of other cryptographic parameters of S-boxes [7], or those of pseudo-random generators in stream ciphers [4, 5], or in computing of parameters of Reed–Muller codes [4], etc.

The algebraic degree of a Boolean function is defined by its special representation—in the area of **algebra** and **cryptology** it is known as an *algebraic normal form*. It is not well-known that this representation is considered in the literature from other viewpoints and has many generalizations, for example:

- **The classical (logical)** one—this representation is known as *Zhegalkin polynomial*. The famous Zhegalkin theorem states that for every Boolean function there exists a unique polynomial form of it over the set of Boolean functions $\{x\,.\,y, x \oplus y, \tilde{1}\}$ ($\tilde{1}$ is the constant 1) [11, 14]. From an algebraic point of view, analogous theorems are formulated and proven in [3, 4].
- **Circuit theory, theory of switching functions**, etc.—such a representation is known as: *modulo-2 sum-of-products, Reed–Muller-canonical expansion, positive polarity Reed–Muller form*, etc. [10, 13].

The computing of these representations for a given Boolean function can be done in different ways [3, 4, 13]. The fastest of them are based on multiplication of a special transformation matrix and the truth table vector of the Boolean function. This matrix is the same in the scientific areas mentioned above and so the obtained algorithms and representations are equivalent. However, depending on the area under consideration, they have different names: *ANF Transform (ANFT), fast Möbius (or Moebius) Transform, Zhegalkin Transform, Positive Polarity Reed–Muller Transform (PPRMT)*, etc. Our experience convinced us how important these facts are in searching papers by keywords.

The algorithm for performing the ANFT in its typical form (i. e., operating on bytes, as Algorithm 1 on page 50) is well-known and is given in many sources. However, its bitwise implementation is considered very seldom, for example in [8, 9]. In [8] J. Fuller paid special attention to the need for appropriate optimization techniques which exploit the full processing power and considered some general optimization issues. She discussed two naive version of the algorithm for ANF transformation and commented on how to optimize their bitwise implementation. J. Fuller used 32-bits computer words and a look-up table where the ANFs of all Boolean functions of 3 variables were precomputed and stored. The code of this algorithm in C language was given in Application C; some results for comparison the running times were also shown. Another known source on the topic is [9]. There A. Joux considered some general issues for optimization borne by the recent microprocessors. The ANFT was represented in its usual (bytewise) form by Algorithm 9.6. Its bitwise implementation in C language was given in Program 9.2. It also operated with 32-bits computer words, but it did not use precomputed ANFs. Firstly the program performed the ANFT on whole computer words (this part is analogous to the one in J. Fuller's work). Thereafter it performed the transform into the computer words—in a top-down manner, from the largest to the smallest size of blocks, conversely to J. Fuller's program. She used left and right shifts, whereas A. Joux used left shifts only. Experimental results about his Program 9.2 were not given.

In both sources [8, 9] the role of shifts and masks was not explained, the correctness and complexity of the programs were not given. So it is not clear how to rewrite them to run on 64-bit (or more bits) computer words.

In 1998 we developed an algorithm for computing the Zhegalkin transform [12]. In 2008 we proposed another version of this algorithm (called PPRMT), which is derived in a different way [1]. We also discussed its bitwise representation and implementation, which can improve its time and space complexity. However, this work stayed out of the attention of cryptographers and a reason for this may have been its name, PPRMT.

This paper is a continuation of [1]. Here we present our full version of the bitwise ANFT with comments about its implementation and correctness. In Section 2 we give the necessary basic notions and offer some preliminary results—a starting point for the next section. At the beginning of it we argue why a bitwise implementation of the ANFT is possible. After that we use an example with a Boolean function of three variables to explain the steps of the bitwise performance of ANFT for the considered function (i. e., for one byte). Then we extend and generalize the conclusions from the example for more variables. The obtained program for performing of the ANFT has a general time complexity of $\Theta((n+44)\,.\,2^{n-7})$ and a space complexity of $\Theta(2^{n-6})$ when we use a bitwise representation of Boolean functions in 64-bit computer words. The code of the program (in C language) and comments about its correctness are given at the end of Section 3. The last section is devoted to the experimental results. Firstly we discuss the conditions for testing and the parameters of the tests. Then the experimental results are presented—they show that the bitwise ANFT is more than 25 times faster in comparison to the usual byte-wise ANFT. Finally some concluding remarks are given.

2. Basic notions and preliminary results.

Here we give the terms and notations usual for cryptography following [4, 3]. We consider the *field of two elements* $\mathbb{F}_2 = \{0, 1\}$ with two operations: sum modulo 2 (XOR), denoted by $x \oplus y$, and multiplication (AND), denoted by $x\,.\,y$ or simply by xy, for $x, y \in \mathbb{F}_2$. The *n-dimensional vector space* over $\mathbb{F}_2$ is $\mathbb{F}_2^n$ and it includes all 2^n binary vectors. For an arbitrary $a = (a_1, a_2, \dots, a_n) \in \mathbb{F}_2^n$, $\bar{a}$ denotes the natural number $\bar{a} = \sum_{i=1}^{n} a_i\,.\,2^{n-i}$ and is often called a *serial number* of the vector a. The binary representation of $\bar{a}$ $(0 \leq \bar{a} \leq 2^n - 1)$ in n bits determines the coordinates of a. The correspondence between a and $\bar{a}$ is a bijection. For $a = (a_1, a_2, \dots, a_n)$ and $b = (b_1, b_2, \dots, b_n) \in \mathbb{F}_2^n$, we say that "*a lexicographically precedes b*" and denote it by $a \prec b$ if $\exists\, k$, $0 \leq k < n$, such that $a_i = b_i$, for $i \leq k$, and $a_{k+1} < b_{k+1}$. The relation "$\prec$", defined over $\mathbb{F}_2^n$, gives a unique *lexicographic* (*standard*) *order* of the vectors of $\mathbb{F}_2^n$ in the sequence $a_0 = (0, 0, \dots, 0) \prec a_1 = (0, 0, \dots, 0, 1) \prec \cdots \prec a_{2^n-1} = (1, 1, \dots, 1)$ and so $\bar{a}_0 = 0 < \bar{a}_1 = 1 < \cdots < \bar{a}_{2^n-1} = 2^n - 1$.

A *Boolean function* of n variables (denoted usually by $x_1, x_2, \dots, x_n$) is a mapping $f : \mathbb{F}_2^n \rightarrow \mathbb{F}_2$, i. e., f maps any binary input $x = (x_1, x_2, \dots, x_n) \in \mathbb{F}_2^n$ to a single binary output $y = f(x) \in \mathbb{F}_2$. Any Boolean function f can be represented in a unique way by the vector of its functional values, called a *Truth Table* vector: $TT(f) = (f_0, f_1, \dots f_{2^n-1})$, where $f_i = f(a_i)$ and a_i is the ith vector in the lexicographic order of $\mathbb{F}_2^n$, for $i = 0, 1, \dots, 2^n - 1$. When the $TT(f)$ is considered as a vector-column, it is denoted by $[f]$. The set of all Boolean functions of n variables is denoted by $\mathcal{F}_n$ and its size is $|\mathcal{F}_n| = 2^{2^n}$.

Another unique representation of any Boolean function $f \in \mathcal{F}_n$ is by *algebraic normal form*, which is a multivariate polynomial

$$f(x_1, x_2, \dots, x_n) = \bigoplus_{u \in \mathbb{F}_2^n} a_{\bar{u}}\, x^u\,. \tag{1}$$

Here $u = (u_1, u_2, \dots, u_n) \in \mathbb{F}_2^n$, $a_{\bar{u}} \in \{0, 1\}$, and x^u means the monomial $x_1^{u_1} x_2^{u_2} \dots x_n^{u_n} = \prod_{i=1}^{n} x_i^{u_i}$, where $x_i^0 = 1$ and $x_i^1 = x_i$, for $i = 1, 2, \dots n$.

When $f \in \mathcal{F}_n$ and the $TT(f)$ (with 2^n values) is given, the values of the coefficients $a_0, a_1, \dots, a_{2^n-1}$ can be computed by a fast algorithm, usually called *ANF transform.* This algorithm is derived in different ways, but its versions are similar to each other. The vector $a = (a_0, a_1, \dots, a_{2^n-1})$ obtained after ANFT is denoted by A_f, or by $[A_f]$ when it is considered as a vector-column.

In [12, 1] we developed an algorithm for fast computing of ANFT in two different ways, based on the equalities with the transformation matrix, as in [10]. So, if $f \in \mathcal{F}_n$ and $TT(f) = (f_0, f_1, \dots f_{2^n-1})$, then:

$$[A_f] = M_n \,.\, [f]\,, \quad \text{and} \quad [f] = M_n^{-1} \,.\, [A_f] \quad \text{over } \mathbb{F}_2.$$

The matrix M_n is defined recursively, as well as by Kronecker product:

$$M_1 = \begin{pmatrix} 1\,0 \\ 1\,1 \end{pmatrix}, \quad M_n = \begin{pmatrix} M_{n-1} & O_{n-1} \\ M_{n-1} & M_{n-1} \end{pmatrix}, \text{ or } M_n = M_1 \otimes M_{n-1} = \bigotimes_{i=1}^{n} M_1,$$

where M_{n-1} is the corresponding transformation matrix of dimension $2^{n-1} \times 2^{n-1}$, and O_{n-1} is a $2^{n-1} \times 2^{n-1}$ zero matrix. Furthermore $M_n = M_n^{-1}$ over $\mathbb{F}_2$ and hence the forward and the inverse ANFT are performed in the same way—so we consider only the forward ANFT. The pseudocode of our algorithm (referred here as Algorithm 1) is given below. The elements of the array f are bytes and they are numbered starting from 0. Every value of $TT(f)$ is stored in a separate byte as a Boolean value true or false. The result A_f is obtained in the same array. The precise estimation of the time complexity of Algorithm 1 is $\Theta(n \,.\, 2^{n-1})$, whereas many authors give an estimation $O(n.2^n)$ (instead of Θ) for analogous algorithms.

Algorithm 1 ANF_Transform (f, n)

```
$blocksize \leftarrow 1$
for $step = 1$ to $n$ do
    $source \leftarrow 0$ {initial position of the first source block}
    while $source < 2^n$ do
        $target \leftarrow source + blocksize$ {initial position of the first target block}
        for $i = 0$ to $blocksize - 1$ do
            $f[target + i] \leftarrow f[target + i]\ XOR\ f[source + i]$
        end for
        $source \leftarrow source + 2 * blocksize$ {beginning of the next source block}
    end while
    $blocksize \leftarrow 2 * blocksize$
end for
return $f$
```

3. Fast implementation of the ANFT.

We note that an implementation of ANFT based on the bitwise representation of Boolean function is possible because:

1. The values of the vectors $TT(f)$ and A_f are zeros or ones, i. e., bits.
2. The sizes of the blocks XOR-ed to other blocks in Algorithm 1 are powers of 2, as well as the sizes of the computer words for representation of integers. So the source and the target blocks occupy parts of a computer word or adjacent computer words—their number is also a power of 2.
3. All necessary operations are available as fast processor instructions—shifts, bitwise XOR, AND, etc.

We note that the well-known Walsh Transform is performed by a similar algorithm. Condition 3 holds for it, whereas conditions 1 and 2 do not, since XOR in condition 2 is replaced by addition of integers and so the resulting vector contains integers. That is why the Walsh Transform cannot have a bitwise implementation.

The idea of a bitwise implementation of the ANFT was given in [1]. Firstly we shall explain it in detail by an example.

Example 1. Let us consider the function $f(x_1, x_2, x_3) = (1, 0, 1, 1, 0, 1, 1, 0)$. The steps of ANFT for f, performed by Algorithn 1, are illustrated as usually by its *butterfly* (*signal-flow*) *diagram* in Figure 1.

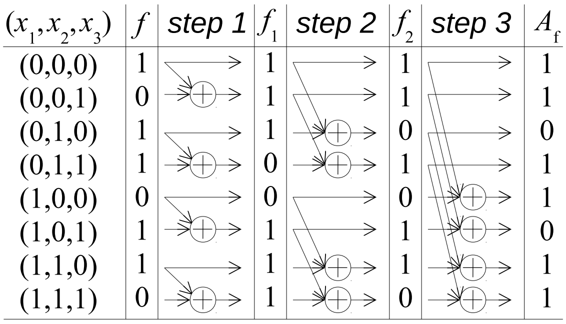

Fig. 1. The butterfly diagram for $f(x_1, x_2, x_3) = (1, 0, 1, 1, 0, 1, 1, 0)$

The bitwise representation of f needs one byte, $f = 10110110$. We follow the steps of Algorithm 1 and implement them in a bitwise manner. We form a mask for each step, so that it has ones in the positions of the source bits and zeros in the remaining (target) bits. For the first step we use the mask $m_1 = 10101010$ (its hexadecimal value is 0XAA)—the values and the operations on each step are given in Table 1, where SHR means a right shift. The assignment $temp = f$ AND m_1 moves the values of all source bits to the variable *temp*, so $temp = 10100010$. A right-shift on *temp* by one bit (since the variable *blocksize*

Table 1. Step-by-step bitwise ANFT for $f(x_1, x_2, x_3) = (1, 0, 1, 1, 0, 1, 1, 0)$

Step	**Variables or operators**	**Values or results**
Input	f	10110110
1.0	m_1	10101010
1.1	$temp = f$ AND m_1	10100010
1.2	$temp = temp$ SHR 1	01010001
1.3	$f = f$ XOR $temp$	11100111
2.0	m_2	11001100
2.1	$temp = f$ AND m_2	11000100
2.2	$temp = temp$ SHR 2	00110001
2.3	$f = f$ XOR $temp$	11010110
3.1	$temp = f$	11010110
3.2	$temp = temp$ SHR 4	00001101
3.3	$f = f$ XOR $temp$	11011011
Output	$A_f = f$	11011011

in Algorithm 1 takes an initial value of 1) yields *temp* $= 01010001$. Thus in *temp*: (1) the source-bits are slid to the positions of the target bits and (2) the places of all source bits in *temp* are occupied by zeros. The third operation is $f = f$ XOR *temp*—thus the values from the source blocks are added modulo 2 to the corresponding values of the target blocks, and the source bits are preserved. So, after the first step f gets the value $f = 11100111$ (as f_1 in Figure 1). In the second step we use the mask $m_2 = 11001100$ (i. e., 0XCC) and right-shift by two bits (the variable *blocksize* in Algorithm 1 takes a value of 2 in the second step), the third operation is the same. Analogously, in the third step the mask is $m_3 = 11110000$ (i. e., 0XF0) and the right-shift is by four bits. This is the half of eight bits and now we have one source block and one target block. So we can obtain the same result in the last step without using the mask m_3. Instead of *temp* $= f$ AND m_3, we simply set *temp* $= f$. After the right shift on *temp* by four bits the result is the same. The last operation is $f = f$ XOR *temp* again. Thus we economize one AND operation. Finally we obtain $A_f = (1, 1, 0, 1, 1, 0, 1, 1)$. Following Eq. (1), the ANF of f is $f(x_1, x_2, x_3) = 1 \oplus x_3 \oplus x_2x_3 \oplus x_1 \oplus x_1x_2 \oplus x_1x_2x_3$.

So, for any $f \in \mathcal{F}_3$, the bitwise ANFT performs the same steps and we unify them in the following three operators in C language:

```
f^= (f & m1) >> 1;   f^= (f & m2) >> 2;   f^= f >> 4;
```

Hence, for any $f \in \mathcal{F}_3$, the bitwise ANFT is performed by $4 + 4 + 3 = 11$ operations.

For the functions $f \in \mathcal{F}_4$, the bitwise ANFT can be done in a similar way. We have to double the masks (i. e., to concatenate each of them with itself) to fill in 2 bytes—so they should be: $m_1 = 1010101010101010$ (0XAAAA) and $m_2 = 1100110011001100$ (0XCCCC). For the third step the mask should be $m_3 = 1111000011110000$ (0XF0F0). For the last (fourth) step we do not need a mask again. Analogously, for the functions $f \in \mathcal{F}_5$ we use four bytes, we double the masks m_1, m_2, m_3 and add a new mask $m_4 =$ 0XFF00FF00, and so on. For the functions $f \in \mathcal{F}_6$ the masks and the steps are given in Listing 1. As we have shown, when $f \in \mathcal{F}_n$ and $3 \leq n \leq 6$, there are n steps. The first $(n - 1)$ of them are performed by four operations and the last one is performed by three operations. So A_f is obtained by $4n - 1$ operations only. Thus we have described the ***first case*** when the bitwise representation of f occupies one computer word of $1, 2, 4, 8$ bytes. So the space complexity is the constant 1 and the time complexity is linear: $\Theta(4n - 1)$.

The ***second case*** is when $f \in \mathcal{F}_n$ and $n > 6$. We use a bitwise representation of f as an array of 2^{n-6} computer words of size $2^6 = 64$ bits. So the space complexity is $\Theta(2^{n-6})$. The **bitwise ANFT** performs two main steps:

Step 1. Bitwise ANFT on each computer word, as was described above. Since we have 2 additional assignments (see Listing 1), we obtain exactly $4.6-1+2 = 25$ operations for one computer word. We have 2^{n-6} computer words and so on this step $\Theta(25\,.\,2^{n-6})$ operations are performed.

Step 2. The usual ANFT (as in Algorithm 1), but with bitwise XORs, performed on whole computer words. The number of operations is $\Theta((n-6)\,.\,2^{n-7})$ on this step.

Hence, for $f \in \mathcal{F}_n$ and $n > 6$, the total time complexity of the bitwise ANFT is

$$\Theta(25\,.\,2^{n-6}) + \Theta((n-6)\,.\,2^{n-7}) = \Theta((n+44)\,.\,2^{n-7})\,.$$

The correctness of the bitwise ANFT follows from the notes at the beginning of this section and also from the correctness of Algorithm 1, since the bitwise ANFT does the same, but in an optimized way. Its code in C language is shown below.

Listing 1. The C code of the bitwise ANFT

```
typedef unsigned long long ull;
const ull m1= 0XAAAAAAAAAAAAAAAA,
          m2= 0XCCCCCCCCCCCCCCCC,
          m3= 0XF0F0F0F0F0F0F0F0,
          m4= 0XFF00FF00FF00FF00,
          m5= 0XFFFF0000FFFF0000;
const int num_of_vars= 8; //number of variables
const int num_of_steps= num_of_vars - 6; //for Step (2) of ANFT
const int num_of_comp_words= 1<<num_of_steps; //=4 for 8 vars
ull f[num_of_comp_words]; // for representation of the TT(f)

void ANF (ull f[]) {
// Step 1 - bitwise ANF on computer words
    for (int k= 0; k < num_of_comp_words; k++) {
        ull temp= f[k];
        temp ^= (temp & m1)>>1;
        temp ^= (temp & m2)>>2;
        temp ^= (temp & m3)>>4;
        temp ^= (temp & m4)>>8;
        temp ^= (temp & m5)>>16;
        temp ^= temp>>32;
        f[k]= temp;
    }
// Step 2
    int blocksize= 1;
    for (int step= 1; step <= num_of_steps; step++) {
        int source= 0;
        while (source < num_of_comp_words) {
            int target= source + blocksize;
            for (int i= 0; i < blocksize; i++) {
                f[target + i] ^= f[source + i];
            }
            source += 2*blocksize;
        }
        blocksize *= 2;
    }
}
```

4. Experimental results and conclusions.

For the usual byte-wise implementation of ANFT (Algorithm 1) and the bitwise ANFT (the implementation from Listing 1) we obtained time complexities of the same type, $\Theta(n \,.\, 2^n)$. The difference between them is in the constants, hidden in the Θ-notation. To compare the real time complexities of both implementations we conduct a sequence of tests at the following conditions:

1. Hardware parameters—Intel Pentium CPU G4400, 3.3 GHz, 4GB RAM, Samsung SSD 650 120 GB.

2. Software parameters—Windows 10 OS and Code::Blocks 13.12 IDE. All programs are created as 32-bits console applications in C++, built in Release mode and executed without Internet connection.

3. Methodology of testing:

    - all tests are performed three times and their running times are taken in average;
    - for every test the obtained ANFs from the byte-wise ANFT and from the bitwise ANFT are compared for matching;
    - the times for file reading, for converting the input to array of bytes (for the byte-wise ANFT), etc., are excluded;
    - the following running times concern only the executions of the byte-wise ANFT and the bitwise ANFT.

For all 2^{16} Boolean functions of 4 variables the running times of the tested programs are very close to 0 seconds and they cannot be compared.

For all 2^{32} Boolean functions of 5 variables and for 32-bits application, the byte-wise ANFT runs in 313.032 seconds, whereas the bitwise ANFT runs in 3.772 seconds—it is about 83 times faster.

For the other tests we created 4 files with $10^6, 10^7, 10^8$ and 10^9 integers, randomly generated in 64-bit computer words. They were used as input data, i. e., as truth table vectors of functions. The following results concern the largest file, its size is ≈14 GB. The running times of both implementations are given in Table 2. The experimental results show that the bitwise ANFT is at least 20 times faster than the byte-wise ANFT for smaller input files and this ratio grows when the file size increases. As Table 2 shows, the bitwise ANFT is more than 25 times faster than the byte-wise ANFT for the largest test file. We consider that it is worth to built the same programs as 64-bit applications and to repeat the tests, although they need many hours of work. The next possible goal – a parallel implementation of the bitwise ANFT and tests with it – is already achieved in [2]. Comparing the results obtained here (for example, milliseconds for computing the ANF of one function) with these in [2], the reader can choose easily when to use a parallel version instead of a serial (non-parallel) version and vice versa.

Table 2. Experimental results about the executions of the byte-wise ANFT and the bitwise ANFT

	Running times in seconds for:					
Number of variables Number of functions	6 10^9	8 $10^9/4$	10 $10^9/16$	12 $10^9/64$	14 $10^9/256$	16 $10^9/1024$
Byte-wise ANFT	170.804	213.741	249.073	285.577	322.185	362.497
Bitwise ANFT	5.032	8.147	9.605	11.356	12.252	12.763
Ratio	33.944:1	26.237:1	25.931:1	25.148:1	26.297:1	28.462:1

Finally, as we mentioned at the beginning of this work, we comment on the numerical results from Section 4 of [6]. There the authors write: “For n = 13; 14 we have not enough memory in our computer to obtain the ANF” (here n is the number of variables). This section (Table 2) and the results from [2] show why this result is very questionable for us. Obviously the implementation of the ANFT in [6] is not efficient enough and hence the comparison results are not realistic. So the necessity of future investigations, estimations, comparisons, etc., between both approaches for computing the algebraic degree of S-boxes becomes evident.

REFERENCES

[1] Bakoev V., K. Manev. Fast computing of the positive polarity Reed–Muller transform over GF(2) and GF(3). In: Proc. of the XI Intern. Workshop on Algebraic and Combinatorial Coding Theory (ACCT), Pamporovo, Bulgaria, 2008, 13–21.

[2] Bikov D., I. Bouyukliev. Parallel Fast Möbius (Reed–Muller) Transform and its Implementation with CUDA on GPUs. In: PASCO 2017, Proc. of the Intern. Workshop on Parallel Symbolic Computation, Kaiserslautern, Germany, 2017, 5:1–5:6.

[3] Canteaut A. Lecture notes on Cryptographic Boolean Functions. Inria, Paris, France, 2016.

[4] Carlet C. Boolean Functions for Cryptography and Error Correcting Codes. In: Crama Y., P. L. Hammer (eds). Boolean Models and Methods in Mathematics, Computer Science, and Engineering. Cambridge Univ. Press, 2010, 257–397.

[5] Carlet C. Vectorial Boolean Functions for Cryptography. In: Crama Y., P. L. Hammer (eds). Boolean Models and Methods in Mathematics, Computer Science, and Engineering. Cambridge Univ. Press, 2010, 398–469.

[6] Climent J.-J., F. García, V. Requena. The degree of a Boolean function and some algebraic properties of its support. In: Data Management and Security, WIT Press, 2013, 25–36.

[7] Çalik Ç. Computing Cryptographic Properties of Boolean Functions from the Algebraic Normal Form Representation. PhD thesis, Middle East Technical University, Ankara, Turkey, 2013.

[8] Fuller J. Analysis of affine equivalent Boolean functions for cryptography. PhD thesis, Queensland University of Technology, Australia, 2003.

[9] Joux A. Algorithmic Cryptanalysis. Chapman & Hall/CRC Cryptography and Network Security, 2012.

[10] Harking B. Efficient algorithm for canonical Reed–Muller expansions of Boolean functions. *IEE Proc. Comput. Digit. Tech.*, **137** (1990), No 5, 366–370.

[11] Manev K. Introduction to Discrete Mathematics. KLMN, Sofia, Bulgaria, 2007 (in Bulgarian).

[12] Manev K., V. Bakoev. Algorithms for performing the Zhegalkin transformation. In: Mathematics and education in mathematics. Proc. of the XXVII Spring Conf. of the Union of Bulgarian Mathematicians (Pleven, Bulgaria, April 9–11, 1998), Sofia, 1998, 229–233.

[13] Porwik P. Efficient calculation of the Reed–Muller form by means of the Walsh transform. *Intern. Journal Appl. Math. Comput. Sci.*, **12** (2002), No 4, 571–579.

[14] Yablonski S. Introduction to Discrete Mathematics. Vysshaia Shkola, Moscow, 2003 (in Russian).

Valentin Bakoev
Faculty of Mathematics and Informatics
University of Veliko Turnovo
Veliko Turnovo, Bulgaria
e-mail: `v.bakoev@uni-vt.bg`

Received August 31, 2017
Final Accepted October 4, 2017